Tcl8.6.14/Tk8.6.14 Documentation > Tcl Commands, version 8.6.14 > interp

Tcl/Tk Applications | Tcl Commands | Tk Commands | [incr Tcl] Package Commands | SQLite3 Package Commands | TDBC Package Commands | tdbc::mysql Package Commands | tdbc::odbc Package Commands | tdbc::postgres Package Commands | tdbc::sqlite3 Package Commands | Thread Package Commands | Tcl C API | Tk C API | [incr Tcl] Package C API | TDBC Package C API

NAME

interp — Create and manipulate Tcl interpreters

SYNOPSIS

DESCRIPTION
THE INTERP COMMAND

interp alias *srcPath srcToken*
interp alias *srcPath srcToken* **{}**
interp alias *srcPath srcCmd targetPath targetCmd* ?*arg arg ...*?
interp aliases ?*path*?
interp bgerror *path* ?*cmdPrefix*?
interp cancel ?**-unwind**? ?**--**? ?*path*? ?*result*?
interp create ?**-safe**? ?**--**? ?*path*?
interp debug *path* ?**-frame** ?*bool*??
interp delete ?*path ...*?
interp eval *path arg* ?*arg ...*?
interp exists *path*
interp expose *path hiddenName* ?*exposedCmdName*?
interp hide *path exposedCmdName* ?*hiddenCmdName*?
interp hidden *path*
interp invokehidden *path* ?*-option ...*? *hiddenCmdName* ?*arg ...*?
interp issafe ?*path*?
interp limit *path limitType* ?*-option*? ?*value ...*?
interp marktrusted *path*
interp recursionlimit *path* ?*newlimit*?
interp share *srcPath channelId destPath*
interp slaves ?*path*?
interp children ?*path*?
interp target *path alias*
interp transfer *srcPath channelId destPath*

child COMMAND

child **aliases**
child **alias** *srcToken*
child **alias** *srcToken* **{}**
child **alias** *srcCmd targetCmd* ?*arg ..*?
child **bgerror** ?*cmdPrefix*?
child **eval** *arg* ?*arg ..*?
child **expose** *hiddenName* ?*exposedCmdName*?
child **hide** *exposedCmdName* ?*hiddenCmdName*?
child **hidden**
child **invokehidden** ?*-option ...*? *hiddenName* ?*arg ..*?
child **issafe**
child **limit** *limitType* ?*-option*? ?*value ...*?
child **marktrusted**
child **recursionlimit** ?*newlimit*?

SAFE INTERPRETERS
ALIAS INVOCATION
HIDDEN COMMANDS
RESOURCE LIMITS

LIMIT OPTIONS

- **-command**
- **-granularity**
- **-milliseconds**
- **-seconds**
- **-value**

BACKGROUND EXCEPTION HANDLING
CREDITS
EXAMPLES
SEE ALSO
KEYWORDS

NAME

interp — Create and manipulate Tcl interpreters

SYNOPSIS

interp *subcommand* ?*arg arg ...*?

DESCRIPTION

This command makes it possible to create one or more new Tcl interpreters that co-exist with the creating interpreter in the same application. The creating interpreter is called the *parent* and the new interpreter is called a *child*. A parent can create any number of children, and each child can itself create additional children for which it is parent, resulting in a hierarchy of interpreters.

Each interpreter is independent from the others: it has its own name space for commands, procedures, and global variables. A parent interpreter may create connections between its children and itself using a mechanism called an *alias*. An *alias* is a command in a child interpreter which, when invoked, causes a command to be invoked in its parent interpreter or in another child interpreter. The only other connections between interpreters are through environment variables (the **env** variable), which are normally shared among all interpreters in the application, and by resource limit exceeded callbacks. Note that the name space for files (such as the names returned by the **open** command) is no longer shared between interpreters. Explicit commands are provided to share files and to transfer references to open files from one interpreter to another.

The **interp** command also provides support for *safe* interpreters. A safe interpreter is a child whose functions have been greatly restricted, so that it is safe to execute untrusted scripts without fear of them damaging other interpreters or the application's environment. For example, all IO channel creation commands and subprocess creation commands are made inaccessible to safe interpreters. See **SAFE INTERPRETERS** below for more information on what features are present in a safe interpreter. The dangerous functionality is not removed from the safe interpreter; instead, it is *hidden*, so that only trusted interpreters can obtain access to it. For a detailed explanation of hidden commands, see **HIDDEN COMMANDS**, below. The alias mechanism can be used for protected communication (analogous to a kernel call) between a child interpreter and its parent. See **ALIAS INVOCATION**, below, for more details on how the alias mechanism works.

A qualified interpreter name is a proper Tcl list containing a subset of its ancestors in the interpreter hierarchy, terminated by the string naming the interpreter in its immediate parent. Interpreter names are relative to the interpreter in which they are used. For example, if "**a**" is a child of the current interpreter and it has a child "**a1**", which in turn has a child "**a11**", the qualified name of "**a11**" in "**a**" is the list "**a1 a11**".

The **interp** command, described below, accepts qualified interpreter names as arguments; the interpreter in which the command is being evaluated can always be referred to as **{}** (the empty list or string). Note that it is impossible to refer to a parent (ancestor) interpreter by name in a child interpreter except through aliases. Also, there is no global name by which one can refer to the first interpreter created in an application. Both restrictions are motivated by safety concerns.

THE INTERP COMMAND

The **interp** command is used to create, delete, and manipulate child interpreters, and to share or transfer channels between interpreters. It can have any of several forms, depending on the *subcommand* argument:

interp alias *srcPath srcToken*
Returns a Tcl list whose elements are the *targetCmd* and *arg*s associated with the alias represented by *srcToken* (this is the value returned when the alias was created; it is possible that the name of the source command in the child is different from

srcToken).

interp alias *srcPath srcToken* **{}**

Deletes the alias for *srcToken* in the child interpreter identified by *srcPath*. *srcToken* refers to the value returned when the alias was created; if the source command has been renamed, the renamed command will be deleted.

interp alias *srcPath srcCmd targetPath targetCmd* ?*arg arg ...*?

This command creates an alias between one child and another (see the **alias** child command below for creating aliases between a child and its parent). In this command, either of the child interpreters may be anywhere in the hierarchy of interpreters under the interpreter invoking the command. *SrcPath* and *srcCmd* identify the source of the alias. *SrcPath* is a Tcl list whose elements select a particular interpreter. For example, "**a b**" identifies an interpreter "**b**", which is a child of interpreter "**a**", which is a child of the invoking interpreter. An empty list specifies the interpreter invoking the command. *srcCmd* gives the name of a new command, which will be created in the source interpreter. *TargetPath* and *targetCmd* specify a target interpreter and command, and the *arg* arguments, if any, specify additional arguments to *targetCmd* which are prepended to any arguments specified in the invocation of *srcCmd*. *TargetCmd* may be undefined at the time of this call, or it may already exist; it is not created by this command. The alias arranges for the given target command to be invoked in the target interpreter whenever the given source command is invoked in the source interpreter. See **ALIAS INVOCATION** below for more details. The command returns a token that uniquely identifies the command created *srcCmd*, even if the command is renamed afterwards. The token may but does not have to be equal to *srcCmd*.

interp aliases ?*path*?

This command returns a Tcl list of the tokens of all the source commands for aliases defined in the interpreter identified by *path*. The tokens correspond to the values returned when the aliases were created (which may not be the same as the current names of the commands).

interp bgerror *path* ?*cmdPrefix*?

This command either gets or sets the current background exception handler for the interpreter identified by *path*. If *cmdPrefix* is absent, the current background exception handler is returned, and if it is present, it is a list of words (of minimum length one) that describes what to set the interpreter's background exception handler to. See the **BACKGROUND EXCEPTION HANDLING** section for more details.

interp cancel ?**-unwind**? ?**--**? ?*path*? ?*result*?

Cancels the script being evaluated in the interpreter identified by *path*. Without the **-unwind** switch the evaluation stack for the interpreter is unwound until an enclosing catch command is found or there are no further invocations of the interpreter left on the call stack. With the **-unwind** switch the evaluation stack for the interpreter is unwound without regard to any intervening catch command until there are no further invocations of the interpreter left on the call stack. The **--** switch can be used to mark the end of switches; it may be needed if *path* is an unusual value such as **-safe**. If *result* is present, it will be used as the error message string; otherwise, a default error message string will be used.

interp create ?**-safe**? ?**--**? ?*path*?

Creates a child interpreter identified by *path* and a new command, called a *child command*. The name of the child command is the last component of *path*. The new child interpreter and the child command are created in the interpreter identified by the path obtained by removing the last component from *path*. For example, if *path* is **a b c** then a new child interpreter and child command named **c** are created in the interpreter identified by the path **a b**. The child command may be used to manipulate the new interpreter as described below. If *path* is omitted, Tcl creates a unique name of the form **interp***x*, where *x* is an integer, and uses it for the interpreter and the child command. If the **-safe** switch is specified (or if the parent interpreter is a safe interpreter), the new child interpreter will be created as a safe interpreter with limited functionality; otherwise the child will include the full set of Tcl built-in commands and variables. The **--** switch can be used to mark the end of switches; it may be needed if *path* is an unusual value such as **-safe**. The result of the command is the name of the new interpreter. The name of a child interpreter must be unique among all the children for its parent; an error occurs if a child interpreter by the given name already exists in this parent. The initial recursion limit of the child interpreter is set to the current recursion limit of its parent interpreter.

interp debug *path* ?**-frame** ?*bool*??

Controls whether frame-level stack information is captured in the child interpreter identified by *path*. If no arguments are given, option and current setting are returned. If **-frame** is given, the debug setting is set to the given boolean if provided and the current setting is returned. This only affects the output of **info frame**, in that exact frame-level information for command invocation at the bytecode level is only captured with this setting on.

For example, with code like

```
proc mycontrol {... script} {
  ...
  uplevel 1 $script
  ...
}

proc dosomething {...} {
  ...
  mycontrol {
    somecode
  }
}
```

the standard setting will provide a relative line number for the command **somecode** and the relevant frame will be of type **eval**. With frame-debug active on the other hand the tracking extends so far that the system will be able to determine the file and absolute line number of this command, and return a frame of type **source**. This more exact information is paid for with slower execution of all commands.

Note that once it is on, this flag cannot be switched back off: such attempts are silently ignored. This is needed to maintain the consistency of the underlying interpreter's state.

interp delete ?*path ...*?
Deletes zero or more interpreters given by the optional *path* arguments, and for each interpreter, it also deletes its children. The command also deletes the child command for each interpreter deleted. For each *path* argument, if no interpreter by that name exists, the command raises an error.

interp eval *path arg* ?*arg ...*?
This command concatenates all of the *arg* arguments in the same fashion as the **concat** command, then evaluates the resulting string as a Tcl script in the child interpreter identified by *path*. The result of this evaluation (including all **return** options, such as **-errorinfo** and **-errorcode** information, if an error occurs) is returned to the invoking interpreter. Note that the script will be executed in the current context stack frame of the *path* interpreter; this is so that the implementations (in a parent interpreter) of aliases in a child interpreter can execute scripts in the child that find out information about the child's current state and stack frame.

interp exists *path*
Returns **1** if a child interpreter by the specified *path* exists in this parent, **0** otherwise. If *path* is omitted, the invoking interpreter is used.

interp expose *path hiddenName* ?*exposedCmdName*?
Makes the hidden command *hiddenName* exposed, eventually bringing it back under a new *exposedCmdName* name (this name is currently accepted only if it is a valid global name space name without any ::), in the interpreter denoted by *path*. If an exposed command with the targeted name already exists, this command fails. Hidden commands are explained in more detail in **HIDDEN COMMANDS**, below.

interp hide *path exposedCmdName* ?*hiddenCmdName*?
Makes the exposed command *exposedCmdName* hidden, renaming it to the hidden command *hiddenCmdName*, or keeping the same name if *hiddenCmdName* is not given, in the interpreter denoted by *path*. If a hidden command with the targeted name already exists, this command fails. Currently both *exposedCmdName* and *hiddenCmdName* can not contain namespace qualifiers, or an error is raised. Commands to be hidden by **interp hide** are looked up in the global namespace even if the current namespace is not the global one. This prevents children from fooling a parent interpreter into hiding the wrong command, by making the current namespace be different from the global one. Hidden commands are explained in more detail in **HIDDEN COMMANDS**, below.

interp hidden *path*
Returns a list of the names of all hidden commands in the interpreter identified by *path*.

interp invokehidden *path* ?*-option ...*? *hiddenCmdName* ?*arg ...*?
Invokes the hidden command *hiddenCmdName* with the arguments supplied in the interpreter denoted by *path*. No substitutions or evaluation are applied to the arguments. Three *-options* are supported, all of which start with **-**: **-namespace** (which takes a single argument afterwards, *nsName*), **-global**, and **--**. If the **-namespace** flag is present, the hidden command is invoked in the namespace called *nsName* in the target interpreter. If the **-global** flag is present, the hidden command is invoked at the global level in the target interpreter; otherwise it is invoked at the current call frame and can access local variables in that and outer call frames. The **--** flag allows the *hiddenCmdName* argument to start with a "-" character, and is otherwise unnecessary. If both the **-namespace** and **-global** flags are present, the **-namespace** flag is ignored. Note that the hidden command will be executed (by default) in the current context stack frame of the *path* interpreter. Hidden commands are explained in more detail in **HIDDEN COMMANDS**, below.

interp issafe ?*path*?
Returns **1** if the interpreter identified by the specified *path* is safe, **0** otherwise.

interp limit *path limitType* ?*-option*? ?*value ...*?

Sets up, manipulates and queries the configuration of the resource limit *limitType* for the interpreter denoted by *path*. If no *-option* is specified, return the current configuration of the limit. If *-option* is the sole argument, return the value of that option. Otherwise, a list of *-option*/*value* argument pairs must supplied. See **RESOURCE LIMITS** below for a more detailed explanation of what limits and options are supported.

interp marktrusted *path*

Marks the interpreter identified by *path* as trusted. Does not expose the hidden commands. This command can only be invoked from a trusted interpreter. The command has no effect if the interpreter identified by *path* is already trusted.

interp recursionlimit *path* ?*newlimit*?

Returns the maximum allowable nesting depth for the interpreter specified by *path*. If *newlimit* is specified, the interpreter recursion limit will be set so that nesting of more than *newlimit* calls to **Tcl_Eval** and related procedures in that interpreter will return an error. The *newlimit* value is also returned. The *newlimit* value must be a positive integer between 1 and the maximum value of a non-long integer on the platform.

The command sets the maximum size of the Tcl call stack only. It cannot by itself prevent stack overflows on the C stack being used by the application. If your machine has a limit on the size of the C stack, you may get stack overflows before reaching the limit set by the command. If this happens, see if there is a mechanism in your system for increasing the maximum size of the C stack.

interp share *srcPath channelId destPath*

Causes the IO channel identified by *channelId* to become shared between the interpreter identified by *srcPath* and the interpreter identified by *destPath*. Both interpreters have the same permissions on the IO channel. Both interpreters must close it to close the underlying IO channel; IO channels accessible in an interpreter are automatically closed when an interpreter is destroyed.

interp slaves ?*path*?

Returns a Tcl list of the names of all the child interpreters associated with the interpreter identified by *path*. If *path* is omitted, the invoking interpreter is used.

interp children ?*path*?

Synonym for . **interp slaves** ?*path*?

interp target *path alias*

Returns a Tcl list describing the target interpreter for an alias. The alias is specified with an interpreter path and source command name, just as in **interp alias** above. The name of the target interpreter is returned as an interpreter path, relative to the invoking interpreter. If the target interpreter for the alias is the invoking interpreter then an empty list is returned. If the target interpreter for the alias is not the invoking interpreter or one of its descendants then an error is generated. The target command does not have to be defined at the time of this invocation.

interp transfer *srcPath channelId destPath*

Causes the IO channel identified by *channelId* to become available in the interpreter identified by *destPath* and unavailable in the interpreter identified by *srcPath*.

child COMMAND

For each child interpreter created with the **interp** command, a new Tcl command is created in the parent interpreter with the same name as the new interpreter. This command may be used to invoke various operations on the interpreter. It has the following general form:

```
child command ?arg arg ...?
```

child is the name of the interpreter, and *command* and the *arg*s determine the exact behavior of the command. The valid forms of this command are:

child **aliases**

Returns a Tcl list whose elements are the tokens of all the aliases in *child*. The tokens correspond to the values returned when the aliases were created (which may not be the same as the current names of the commands).

child **alias** *srcToken*

Returns a Tcl list whose elements are the *targetCmd* and *arg*s associated with the alias represented by *srcToken* (this is the value returned when the alias was created; it is possible that the actual source command in the child is different from *srcToken*).

child **alias** *srcToken* **{}**

Deletes the alias for *srcToken* in the child interpreter. *srcToken* refers to the value returned when the alias was created; if the source command has been renamed, the renamed command will be deleted.

child **alias** *srcCmd targetCmd* ?*arg* ..?
: Creates an alias such that whenever *srcCmd* is invoked in *child*, *targetCmd* is invoked in the parent. The *arg* arguments will be passed to *targetCmd* as additional arguments, prepended before any arguments passed in the invocation of *srcCmd*. See **ALIAS INVOCATION** below for details. The command returns a token that uniquely identifies the command created *srcCmd*, even if the command is renamed afterwards. The token may but does not have to be equal to *srcCmd*.

child **bgerror** ?*cmdPrefix*?
: This command either gets or sets the current background exception handler for the *child* interpreter. If *cmdPrefix* is absent, the current background exception handler is returned, and if it is present, it is a list of words (of minimum length one) that describes what to set the interpreter's background exception handler to. See the **BACKGROUND EXCEPTION HANDLING** section for more details.

child **eval** *arg* ?*arg* ..?
: This command concatenates all of the *arg* arguments in the same fashion as the **concat** command, then evaluates the resulting string as a Tcl script in *child*. The result of this evaluation (including all **return** options, such as **-errorinfo** and **-errorcode** information, if an error occurs) is returned to the invoking interpreter. Note that the script will be executed in the current context stack frame of *child*; this is so that the implementations (in a parent interpreter) of aliases in a child interpreter can execute scripts in the child that find out information about the child's current state and stack frame.

child **expose** *hiddenName* ?*exposedCmdName*?
: This command exposes the hidden command *hiddenName*, eventually bringing it back under a new *exposedCmdName* name (this name is currently accepted only if it is a valid global name space name without any ::), in *child*. If an exposed command with the targeted name already exists, this command fails. For more details on hidden commands, see **HIDDEN COMMANDS**, below.

child **hide** *exposedCmdName* ?*hiddenCmdName*?
: This command hides the exposed command *exposedCmdName*, renaming it to the hidden command *hiddenCmdName*, or keeping the same name if the argument is not given, in the *child* interpreter. If a hidden command with the targeted name already exists, this command fails. Currently both *exposedCmdName* and *hiddenCmdName* can not contain namespace qualifiers, or an error is raised. Commands to be hidden are looked up in the global namespace even if the current namespace is not the global one. This prevents children from fooling a parent interpreter into hiding the wrong command, by making the current namespace be different from the global one. For more details on hidden commands, see **HIDDEN COMMANDS**, below.

child **hidden**
: Returns a list of the names of all hidden commands in *child*.

child **invokehidden** ?*-option* ...? *hiddenName* ?*arg* ..?
: This command invokes the hidden command *hiddenName* with the supplied arguments, in *child*. No substitutions or evaluations are applied to the arguments. Three *-option*s are supported, all of which start with **-**: **-namespace** (which takes a single argument afterwards, *nsName*), **-global**, and **--**. If the **-namespace** flag is given, the hidden command is invoked in the specified namespace in the child. If the **-global** flag is given, the command is invoked at the global level in the child; otherwise it is invoked at the current call frame and can access local variables in that or outer call frames. The **--** flag allows the *hiddenCmdName* argument to start with a “-” character, and is otherwise unnecessary. If both the **-namespace** and **-global** flags are given, the **-namespace** flag is ignored. Note that the hidden command will be executed (by default) in the current context stack frame of *child*. For more details on hidden commands, see **HIDDEN COMMANDS**, below.

child **issafe**
: Returns **1** if the child interpreter is safe, **0** otherwise.

child **limit** *limitType* ?*-option*? ?*value* ...?
: Sets up, manipulates and queries the configuration of the resource limit *limitType* for the child interpreter. If no *-option* is specified, return the current configuration of the limit. If *-option* is the sole argument, return the value of that option. Otherwise, a list of *-option*/*value* argument pairs must supplied. See **RESOURCE LIMITS** below for a more detailed explanation of what limits and options are supported.

child **marktrusted**
: Marks the child interpreter as trusted. Can only be invoked by a trusted interpreter. This command does not expose any hidden commands in the child interpreter. The command has no effect if the child is already trusted.

child **recursionlimit** ?*newlimit*?
: Returns the maximum allowable nesting depth for the *child* interpreter. If *newlimit* is specified, the recursion limit in *child* will be set so that nesting of more than *newlimit* calls to **Tcl_Eval()** and related procedures in *child* will return an error. The

newlimit value is also returned. The *newlimit* value must be a positive integer between 1 and the maximum value of a non-long integer on the platform.

The command sets the maximum size of the Tcl call stack only. It cannot by itself prevent stack overflows on the C stack being used by the application. If your machine has a limit on the size of the C stack, you may get stack overflows before reaching the limit set by the command. If this happens, see if there is a mechanism in your system for increasing the maximum size of the C stack.

SAFE INTERPRETERS

A safe interpreter is one with restricted functionality, so that is safe to execute an arbitrary script from your worst enemy without fear of that script damaging the enclosing application or the rest of your computing environment. In order to make an interpreter safe, certain commands and variables are removed from the interpreter. For example, commands to create files on disk are removed, and the **exec** command is removed, since it could be used to cause damage through subprocesses. Limited access to these facilities can be provided, by creating aliases to the parent interpreter which check their arguments carefully and provide restricted access to a safe subset of facilities. For example, file creation might be allowed in a particular subdirectory and subprocess invocation might be allowed for a carefully selected and fixed set of programs.

A safe interpreter is created by specifying the **-safe** switch to the **interp create** command. Furthermore, any child created by a safe interpreter will also be safe.

A safe interpreter is created with exactly the following set of built-in commands:

after	**append**	**apply**	**array**
binary	**break**	**catch**	**chan**
clock	**close**	**concat**	**continue**
dict	**eof**	**error**	**eval**
expr	**fblocked**	**fcopy**	**fileevent**
flush	**for**	**foreach**	**format**
gets	**global**	**if**	**incr**
info	**interp**	**join**	**lappend**
lassign	**lindex**	**linsert**	**list**
llength	**lrange**	**lrepeat**	**lreplace**
lsearch	**lset**	**lsort**	**namespace**
package	**pid**	**proc**	**puts**
read	**regexp**	**regsub**	**rename**
return	**scan**	**seek**	**set**
split	**string**	**subst**	**switch**
tell	**time**	**trace**	**unset**
update	**uplevel**	**upvar**	**variable**
vwait	**while**		

The following commands are hidden by **interp create** when it creates a safe interpreter:

cd	**encoding**	**exec**	**exit**
fconfigure	**file**	**glob**	**load**
open	**pwd**	**socket**	**source**
unload			

These commands can be recreated later as Tcl procedures or aliases, or re-exposed by **interp expose**.

The following commands from Tcl's library of support procedures are not present in a safe interpreter:

auto_exec_ok	**auto_import**	**auto_load**
auto_load_index	**auto_qualify**	**unknown**

Note in particular that safe interpreters have no default **unknown** command, so Tcl's default autoloading facilities are not available. Autoload access to Tcl's commands that are normally autoloaded:

auto_mkindex	**auto_mkindex_old**

auto_reset	history
parray	**pkg_mkIndex**
::pkg::create	**::safe::interpAddToAccessPath**
::safe::interpCreate	**::safe::interpConfigure**
::safe::interpDelete	**::safe::interpFindInAccessPath**
::safe::interpInit	**::safe::setLogCmd**
tcl_endOfWord	**tcl_findLibrary**
tcl_startOfNextWord	**tcl_startOfPreviousWord**
tcl_wordBreakAfter	**tcl_wordBreakBefore**

can only be provided by explicit definition of an **unknown** command in the safe interpreter. This will involve exposing the **source** command. This is most easily accomplished by creating the safe interpreter with Tcl's **Safe-Tcl** mechanism. **Safe-Tcl** provides safe versions of **source**, **load**, and other Tcl commands needed to support autoloading of commands and the loading of packages.

In addition, the **env** variable is not present in a safe interpreter, so it cannot share environment variables with other interpreters. The **env** variable poses a security risk, because users can store sensitive information in an environment variable. For example, the PGP manual recommends storing the PGP private key protection password in the environment variable *PGPPASS*. Making this variable available to untrusted code executing in a safe interpreter would incur a security risk.

If extensions are loaded into a safe interpreter, they may also restrict their own functionality to eliminate unsafe commands. For a discussion of management of extensions for safety see the manual entries for **Safe-Tcl** and the **load** Tcl command.

A safe interpreter may not alter the recursion limit of any interpreter, including itself.

ALIAS INVOCATION

The alias mechanism has been carefully designed so that it can be used safely in an untrusted script which is being executed in a safe interpreter even if the target of the alias is not a safe interpreter. The most important thing in guaranteeing safety is to ensure that information passed from the child to the parent is never evaluated or substituted in the parent; if this were to occur, it would enable an evil script in the child to invoke arbitrary functions in the parent, which would compromise security.

When the source for an alias is invoked in the child interpreter, the usual Tcl substitutions are performed when parsing that command. These substitutions are carried out in the source interpreter just as they would be for any other command invoked in that interpreter. The command procedure for the source command takes its arguments and merges them with the *targetCmd* and *arg*s for the alias to create a new array of arguments. If the words of *srcCmd* were “*srcCmd arg1 arg2 ... argN*”, the new set of words will be “*targetCmd arg arg ... arg arg1 arg2 ... argN*”, where *targetCmd* and *arg*s are the values supplied when the alias was created. *TargetCmd* is then used to locate a command procedure in the target interpreter, and that command procedure is invoked with the new set of arguments. An error occurs if there is no command named *targetCmd* in the target interpreter. No additional substitutions are performed on the words: the target command procedure is invoked directly, without going through the normal Tcl evaluation mechanism. Substitutions are thus performed on each word exactly once: *targetCmd* and *args* were substituted when parsing the command that created the alias, and *arg1 - argN* are substituted when the alias's source command is parsed in the source interpreter.

When writing the *targetCmd*s for aliases in safe interpreters, it is very important that the arguments to that command never be evaluated or substituted, since this would provide an escape mechanism whereby the child interpreter could execute arbitrary code in the parent. This in turn would compromise the security of the system.

HIDDEN COMMANDS

Safe interpreters greatly restrict the functionality available to Tcl programs executing within them. Allowing the untrusted Tcl program to have direct access to this functionality is unsafe, because it can be used for a variety of attacks on the environment. However, there are times when there is a legitimate need to use the dangerous functionality in the context of the safe interpreter. For example, sometimes a program must be **source**d into the interpreter. Another example is Tk, where windows are bound to the hierarchy of windows for a specific interpreter; some potentially dangerous functions, e.g. window management, must be performed on these windows within the interpreter context.

The **interp** command provides a solution to this problem in the form of *hidden commands*. Instead of removing the dangerous commands entirely from a safe interpreter, these commands are hidden so they become unavailable to Tcl scripts executing in the interpreter. However, such hidden commands can be invoked by any trusted ancestor of the safe interpreter, in the context of the safe interpreter, using **interp invoke**. Hidden commands and exposed commands reside in separate name spaces. It is possible to define a hidden command and an exposed command by the same name within one interpreter.

Hidden commands in a child interpreter can be invoked in the body of procedures called in the parent during alias invocation. For example, an alias for **source** could be created in a child interpreter. When it is invoked in the child interpreter, a procedure is called in the parent interpreter to check that the operation is allowable (e.g. it asks to source a file that the child interpreter is allowed to access). The procedure then it invokes the hidden **source** command in the child interpreter to actually source in the contents of the file. Note that two commands named **source** exist in the child interpreter: the alias, and the hidden command.

Because a parent interpreter may invoke a hidden command as part of handling an alias invocation, great care must be taken to avoid evaluating any arguments passed in through the alias invocation. Otherwise, malicious child interpreters could cause a trusted parent interpreter to execute dangerous commands on their behalf. See the section on **ALIAS INVOCATION** for a more complete discussion of this topic. To help avoid this problem, no substitutions or evaluations are applied to arguments of **interp invokehidden**.

Safe interpreters are not allowed to invoke hidden commands in themselves or in their descendants. This prevents them from gaining access to hidden functionality in themselves or their descendants.

The set of hidden commands in an interpreter can be manipulated by a trusted interpreter using **interp expose** and **interp hide**. The **interp expose** command moves a hidden command to the set of exposed commands in the interpreter identified by *path*, potentially renaming the command in the process. If an exposed command by the targeted name already exists, the operation fails. Similarly, **interp hide** moves an exposed command to the set of hidden commands in that interpreter. Safe interpreters are not allowed to move commands between the set of hidden and exposed commands, in either themselves or their descendants.

Currently, the names of hidden commands cannot contain namespace qualifiers, and you must first rename a command in a namespace to the global namespace before you can hide it. Commands to be hidden by **interp hide** are looked up in the global namespace even if the current namespace is not the global one. This prevents children from fooling a parent interpreter into hiding the wrong command, by making the current namespace be different from the global one.

RESOURCE LIMITS

Every interpreter has two kinds of resource limits that may be imposed by any parent interpreter upon its children. Command limits (of type **command**) restrict the total number of Tcl commands that may be executed by an interpreter (as can be inspected via the **info cmdcount** command), and time limits (of type **time**) place a limit by which execution within the interpreter must complete. Note that time limits are expressed as *absolute* times (as in **clock seconds**) and not relative times (as in **after**) because they may be modified after creation.

When a limit is exceeded for an interpreter, first any handler callbacks defined by parent interpreters are called. If those callbacks increase or remove the limit, execution within the (previously) limited interpreter continues. If the limit is still in force, an error is generated at that point and normal processing of errors within the interpreter (by the **catch** command) is disabled, so the error propagates outwards (building a stack-trace as it goes) to the point where the limited interpreter was invoked (e.g. by **interp eval**) where it becomes the responsibility of the calling code to catch and handle.

LIMIT OPTIONS

Every limit has a number of options associated with it, some of which are common across all kinds of limits, and others of which are particular to the kind of limit.

-command
: This option (common for all limit types) specifies (if non-empty) a Tcl script to be executed in the global namespace of the interpreter reading and writing the option when the particular limit in the limited interpreter is exceeded. The callback may modify the limit on the interpreter if it wishes the limited interpreter to continue executing. If the callback generates an exception, it is reported through the background exception mechanism (see **BACKGROUND EXCEPTION HANDLING**). Note that the callbacks defined by one interpreter are completely isolated from the callbacks defined by another, and that the order in which those callbacks are called is undefined.

-granularity
: This option (common for all limit types) specifies how frequently (out of the points when the Tcl interpreter is in a consistent state where limit checking is possible) that the limit is actually checked. This allows the tuning of how frequently a limit is checked, and hence how often the limit-checking overhead (which may be substantial in the case of time limits) is incurred.

-milliseconds
: This option specifies the number of milliseconds after the moment defined in the **-seconds** option that the time limit will fire. It should only ever be specified in conjunction with the **-seconds** option (whether it was set previously or is being set this invocation.)

-seconds

This option specifies the number of seconds after the epoch (see **clock seconds**) that the time limit for the interpreter will be triggered. The limit will be triggered at the start of the second unless specified at a sub-second level using the **-milliseconds** option. This option may be the empty string, which indicates that a time limit is not set for the interpreter.

-value

This option specifies the number of commands that the interpreter may execute before triggering the command limit. This option may be the empty string, which indicates that a command limit is not set for the interpreter.

Where an interpreter with a resource limit set on it creates a child interpreter, that child interpreter will have resource limits imposed on it that are at least as restrictive as the limits on the creating parent interpreter. If the parent interpreter of the limited parent wishes to relax these conditions, it should hide the **interp** command in the child and then use aliases and the **interp invokehidden** subcommand to provide such access as it chooses to the **interp** command to the limited parent as necessary.

BACKGROUND EXCEPTION HANDLING

When an exception happens in a situation where it cannot be reported directly up the stack (e.g. when processing events in an **update** or **vwait** call) the exception is instead reported through the background exception handling mechanism. Every interpreter has a background exception handler registered; the default exception handler arranges for the **bgerror** command in the interpreter's global namespace to be called, but other exception handlers may be installed and process background exceptions in substantially different ways.

A background exception handler consists of a non-empty list of words to which will be appended two further words at invocation time. The first word will be the interpreter result at time of the exception, typically an error message, and the second will be the dictionary of return options at the time of the exception. These are the same values that **catch** can capture when it controls script evaluation in a non-background situation. The resulting list will then be executed in the interpreter's global namespace without further substitutions being performed.

CREDITS

The safe interpreter mechanism is based on the Safe-Tcl prototype implemented by Nathaniel Borenstein and Marshall Rose.

EXAMPLES

Creating and using an alias for a command in the current interpreter:

```
interp alias {} getIndex {} lsearch {alpha beta gamma delta}
set idx [getIndex delta]
```

Executing an arbitrary command in a safe interpreter where every invocation of **lappend** is logged:

```
set i [interp create -safe]
interp hide $i lappend
interp alias $i lappend {} loggedLappend $i
proc loggedLappend {i args} {
    puts "logged invocation of lappend $args"
    interp invokehidden $i lappend {*}$args
}
interp eval $i $someUntrustedScript
```

Setting a resource limit on an interpreter so that an infinite loop terminates.

```
set i [interp create]
interp limit $i command -value 1000
interp eval $i {
    set x 0
    while {1} {
        puts "Counting up... [incr x]"
    }
}
```

SEE ALSO

bgerror, **load**, **safe**, **Tcl_CreateChild**, **Tcl_Eval**, **Tcl_BackgroundException**

KEYWORDS

alias, parent interpreter, safe interpreter, child interpreter